Third International Workshop on Analysis of Security APIs

Port Jefferson, Long Island, NY USA

July 10-11 2009

Synthesising Secure APIs

Véronique Cortier
LORIA, Projet Cassis, CNRS & INRIA,
cortier@loria.fr

Graham Steel
LSV, CNRS & ENS de Cachan, France
graham.steel@lsv.ens-cachan.fr

Abstract

Security APIs are used to define the boundary between trusted and untrusted code. The security properties of existing API are not always clear. In this paper, we give a new generic API for managing symmetric keys on a trusted cryptographic device. We state and prove security properties for the API. In particular, our API offers a high level of security even when the host machine is controlled by an attacker.

Our API is generic in the sense that it can implement a wide variety of (symmetric key) protocols. As a proof of concept, we give an algorithm for automatically instantiating the API commands for a given key management protocol. We demonstrate the algorithm on a set of key establishment protocols from the Clark-Jacob suite.

Security APIs are used to define the boundary between trusted and untrusted code. They typically arise in systems where certain security-critical fragments of a programe are executed on some tamper resistant device (TRD), such as a smartcard, USB security token or hardware security module (HSM). Though they typically employ cryptography, security APIs differ from regular cryptographic APIs in that they are designed to enforce a policy, i.e. no matter what API commands are received from the (possibly malicious) untrusted code, certain properties will continue to hold, e.g. the secrecy of sensitive cryptographic keys.

The ability of these APIs to enforce their policies has been the subject of formal and informal analysis in recent years. Open standards such as PKCS#11 [12] and proprietary solutions such as IBM's Common Cryptographic Architecture [3] have been shown to have flaws which may lead to breaches of the policy [2, 5, 6, 8, 10]. The situation is complicated by the lack of a clearly specified security policy, leading to disputes over what does and does not constitute an attack [9]. All this leaves the application developer in a confusing position. Since more and more applications are turning to TRD based solutions for enforcing security [1, 11] there is a pressing need for solutions.

We propose to tackle this problem from a different direction. We suggest a way to infer functional properties of a security API for a TRD from the security protocols the device is supposed to support. Our first main contribution is to give a generic API for key management protocols. Our API is generic in the sense that it can implement a wide class of symmetric key protocols, while ensuring security of confidential data. The key idea is that confidential data should be stored inside a secure component together with the set of agents that are granted access to it. Then our API will encrypt data only if the agents that are granted access to the encryption key are all also granted access to the encrypted data. In this way, trusted data can be securely shared between TRDs. To illustrate the generality of our API, we show how to instantiate the API commands for a given protocol using a simple algorithm that has been implemented in Prolog. In particular, we show that our API supports a suite of well-known key establishment protocols.

Our second main contribution is to state and prove key security properties for the API no matter what protocol has been implemented. We propose a formal model for a threat scenario where TRDs may sometimes be connected to a clean host machine, and sometimes to a corrupted one where the attacker can execute arbitrary code. Additionally, the attacker is assumed to have defeated the tamper resistance on some devices, obtaining the long term keys of some users. We show in particular that our API guarantees the confidentiality of any (non public) data that is meant to be shared between honest agents only (honest agents are those whose TRDs are intact). The property holds even when honest agents

APIs are controlled by an attacker (in case e.g. an honest user's machine has been infected by a worm). Considering an even stronger attack scenario, where the attacker is also given old confidential keys, we show that our API still provides security provided it is switched to a restricted mode where the API decrypts a cyphertext only when it is able to perform some freshness test. This restricted mode allows us to implement fewer protocols. In particular, of course it does not allow us to implement protocols subject to replay attacks. It does not cover all notions of freshness, but in fact, we discovered that any symmetric key establishment protocol of the Clark and Jacob library [4] can be implemented within the restricted mode, except for protocols that are known to suffer from replay attacks.

A detailed version of our contributions can be found online as an INRIA Research Report [7].

References

[1] Council regulation (ec) no 2252/2004: on standards for security features and biometrics in passports and travel documents issued by member states, December 2004. Available at `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2004:385:0001:0006:EN:PDF`.

[2] M. Bond. Attacks on cryptoprocessor transaction sets. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'01)*, volume 2162 of *LNCS*, pages 220–234, Paris, France, 2001. Springer.

[3] *CCA Basic Services Reference and Guide*, Oct. 2006. Available online at `www.ibm.com/security/cryptocards/pdfs/bs327.pdf`.

[4] J. Clark and J. Jacob. A survey of authentication protocol literature: Version 1.0. Available via `http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz`, 1997.

[5] J. Clulow. On the security of PKCS#11. In *Proceedings of the 5th International Worshop on Cryptographic Hardware and Embedded Systems (CHES'03)*, volume 2779 of *LNCS*, pages 411–425, Cologne, Germany, 2003. Springer.

[6] V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, volume 4424 of *LNCS*, pages 538–552, Braga, Portugal, 2007. Springer.

[7] V. Cortier and G. Steel. Synthesising secure APIs. Research Report RR-6882, INRIA, March 2009.

[8] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[9] IBM Comment on "A Chosen Key Difference Attack on Control Vectors", Jan. 2001. Available from `http://www.cl.cam.ac.uk/~mkb23/research.html`.

[10] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

[11] M. Raya and J.-P. Hubaux. Securing vehicular ad hoc networks. *Journal of Computer Security*, 15(1):39–68, 2007.

[12] RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard.*, June 2004.

Strict Access Control in a Key-Management Server

Christian Cachin Anil Kurmus Marko Vukolić
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{cca,kur,mvu}@zurich.ibm.com

3 June 2009

1 Introduction

Key management is concerned with operations to manage the lifecycle of cryptographic keys, for creating, storing, distributing, deploying, and deleting keys. An important aspect is to manage the attributes of keys that govern their usage and their relation to other keys. Multiple efforts are currently underway to build and standardize key-management systems accessible over open networks: the W3C XML Key Management Specification (XKMS) [18], the IEEE P1619.3 Key Management Project [12], the OASIS Key Management Interoperability Protocol (KMIP) standardization effort [14], and the Sun Crypto Key Management System [16] are some of the most prominent ones. Cover [9] gives an up-to-date summary of the current developments.

Many proprietary key-management systems are on the market, including HP StorageWorks Secure Key Manager, IBM Distributed Key Management System (DKMS), IBM Tivoli Key Lifecycle Manager (TKLM), NetApp Lifetime Key Management, and Thales/nCipher keyAuthority. The need for enterprise-wide key management systems has been recognized widely [4], and NIST, an agency of the US Government, has issued a general recommendation for key management [3].

Such a key-management server is generally accessed by multiple clients, who perform operations on keys and other cryptographic objects maintained by the server. The objects may include *symmetric keys*, *public keys*, *private keys*, *certificates*, and more; they typically have a range of attributes describing their lifecycle and their usage in cryptographic operations. Operations allow to create, import, read, search, update, and delete keys by the server, and generally focus on attribute handling rather than on cryptographic functions. A comprehensive key-management server will also support some small set of cryptographic operations, including to create a key, to issue a certificate, to *derive* a new key (a deterministic operation that creates a symmetric key from an existing one), and to *wrap* or *unwrap* a key with another key (wrapping means to encrypt a target key with another key for export and transfer to another system). These features can be found in many of the above-mentioned protocols and systems.

In this work, we report on a design for controlling access to operations and to keys in a key-server prototype, which we are currently developing. The key server is able to distinguish between different *users*, which are the principals that invoke operations, and to securely authenticate them.

Because the key-management server provides the above-mentioned cryptographic functions, it represents a *cryptographic security API* accessible over a network. Security APIs stand at the boundary between untrusted code and trusted modules capable of maintaining internal state. Cryptographic security APIs are typically provided by cryptographic tokens [1], hardware-security modules (HSM) like IBM's 4764 cryptoprocessor that supports the IBM CCA interface [11, 13] and generic PKCS #11-compliant [15] modules, smartcards, or the Trusted Platform Module [17]. This work extends the study of cryptographic security APIs to protocols over open networks.

2 Access Control

We distinguish between *basic* and *strict* access control in the key server. In *basic* mode, access-control decisions for a key are taken directly from an *access-control list (ACL)* associated with it. But because the operations of our key server allow users to create complex relationships between keys, primarily through key derivation and key wrapping, basic access control may have security problems. For example, if there exists a particular key that some user is not allowed to read, but the user may wrap that key under another key of its choice and export the wrapped representation, the user may nevertheless obtain the bits of the first key. Another example is a key that was derived from a parent key by the server; when a user reads the parent key, the user implicitly also obtains the cryptographic material of the derived key.

In general, a cryptographic interface that manages keys and allows to create such dependencies among keys poses the problem that access to one key may give a user access to many another keys. This issue has been identified in the APIs of several cryptographic modules [2, 6, 8, 10] and may lead to serious security breaches when an organization does not fully understand all implications of an API.

In *strict* mode, therefore, access-control decisions by the key server take the semantics of the key-management API into account and implement a cryptographically sound access-control policy on all symmetric keys and private keys. The above issues with basic access control are eliminated with strict access control. Our strict access-control policy builds on the work of Cachin and Chandran [7], which describes a secure cryptographic token interface, introduces a cryptographically strong security policy, and shows how to implement it. A strict access-control decision may not only depend on the ACL of the corresponding key, but takes also into account the ACLs of related keys and the history of past operations executed on them. It prevents any unauthorized disclosure of a symmetric key or a private key.

Every key maintained by the server has several *attributes* that govern if an access is permitted. The basic policy is determined by an *access-control list (ACL)* attribute. It can be modified by clients and contains a list of user/privilege-pairs. A boolean attribute *strict* determines if the key underlies only the basic or the strict access-control policy.

Every key maintained by the key-management server in strict mode benefits from an explicitly stated security policy that respects cryptographic side-effects of the server's operations. In particular, it guarantees that a user may only retrieve the information she is authorized to, i.e., that she cannot abuse the API to violate the access control policy. Thus, it avoids the problems mentioned above and similar problems existing in other APIs [2, 6, 8, 10], which arise from interdependencies among the keys.

In a forthcoming paper [5], we report on the challenges with designing and on the lessons learned from implementing strict access control in the prototype key-management server.

References

[1] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov, "Cryptographic processors — a survey," *Proceedings of the IEEE*, vol. 94, pp. 357–369, Feb. 2006.

[2] R. J. Anderson, "Why cryptosystems fail," in *Proc. 1st ACM Conference on Computer and Communications Security (CCS)*, pp. 215–227, 1993.

[3] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management," NIST special publication 800-57, National Institute of Standards and Technology (NIST), 2007. Available from `http://csrc.nist.gov/publications/PubsSPs.html`.

[4] BITS Security Working Group, "Enterprise key management." Whitepaper, BITS Financial Services Roundtable, available from `http://www.bits.org/downloads/Publications%20Page/BITSEnterpriseKeyManagementMay2008.pdf`, May 2008.

[5] M. Björkqvist, C. Cachin, R. Haas, X.-Y. Hu, A. Kurmus, R. Pawlitzek, and M. Vukolić, "Design and implementation of a key-lifecycle management system," Research Report RZ 3739, IBM Research, June 2009.

[6] M. Bond, "Attacks on cryptoprocessor transaction sets," in *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2162 of *Lecture Notes in Computer Science*, pp. 220–234, 2001.

[7] C. Cachin and N. Chandran, "A secure cryptographic token interface," in *Proc. Computer Security Foundations Symposium (CSF-22)*, IEEE, July 2009. To appear.

[8] J. Clulow, "On the security of PKCS#11," in *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 411–425, 2003.

[9] "Cover pages: Cryptographic key management." `http://xml.coverpages.org/keyManagement.html`, Apr. 2009.

[10] S. Delaune, S. Kremer, and G. Steel, "Formal analysis of PKCS#11," in *Proc. 21st IEEE Computer Security Foundations Symposium (CSF)*, 2008.

[11] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 secure coprocessor," *IEEE Computer*, vol. 34, pp. 57–66, Oct. 2001.

[12] IEEE Security in Storage Working Group (SISWG), "P1619.3/D6 draft standard for key management infrastructure for cryptographic protection of stored data." Available from `https://siswg.net/index.php`, 2009.

[13] International Business Machines Corp., *CCA Basic Services Reference and Guide for the IBM 4758 PCI and IBM 4764 PCI-X Cryptographic Coprocessors*, 19th ed., Sept. 2008. Available from `http://www-03.ibm.com/security/cryptocards/pcicc/library.shtml`.

[14] OASIS Key Management Interoperability Protocol Technical Committee, "Key Management Interoperability Protocol," Apr. 2009. Editor's draft 0.98; available from `http://www.oasis-open.org/committees/documents.php?wg_abbrev=kmip`.

[15] RSA Laboratories, "PKCS #11 v2.20: Cryptographic Token Interface Standard." Available from `http://www.rsa.com/rsalabs/`, 2004.

[16] Sun Microsystems, "Sun Crypto Key Management System (KMS)." `http://opensolaris.org/os/project/kmsagenttoolkit/`, 2009.

[17] Trusted Computing Group, "Trusted platform module specifications." Available from `http://www.trustedcomputinggroup.org`, 2008.

[18] World Wide Web Consortium, XML Key Management Working Group, "XML Key Management Specification (XKMS 2.0)." Available from `http://www.w3.org/2001/XKMS/`, 2005.

Security APIs for Privacy in Social Networks

Jonathan Anderson
Computer Laboratory
University of Cambridge
jonathan.anderson@ieee.org

Joseph Bonneau
Computer Laboratory
University of Cambridge
joseph.bonneau@cl.cam.ac.uk

Frank Stajano
Computer Laboratory
University of Cambridge
frank.stajano@cl.cam.ac.uk

I. Introduction

Online social networks, in their current form, require users to place a vast amount of trust in the operators of both the core network and the third-party applications they use. Since both of these actors have shown themselves to be untrustworthy in the past [1], [2], [3], [4], [5], we have proposed a model for social networks in which client software runs on the user's computer, encrypted blocks are stored on a "dumb" server and third-party applications are sandboxed to avoid the leakage of personal information [6].

In this scheme, the interface between applications and the core client software resembles a system call API in which a kernel offers applications the means to perform privileged operations. We have begun exploring this API to determine its functional requirements and desired security properties, but we welcome comments from and engagement with the security API community in order to provide the users of social networks with meaningful promises of personal privacy.

II. Functional Requirements

An API for social applications must allow users to adapt the technology to their own needs [7]; we cannot predict what applications users will find compelling. We must, however, build this API so that it can support the functionality that we already know users will expect. This functionality includes finding other users, especially real-life friends, communicating and sharing personal information with them and adding value to other users' content by tagging and commenting on it.

a) Finding Friends: In order to establish connections, a member of a social network must advertise themselves to other users. By default, users of a site should be invisible to all other users. They should be able to advertise themselves in three ways:

1) By making some personal information (e.g. name, photo) available on the Web;
2) By sending social network information via existing communication channels (e.g. IM, e-mail);
3) Via social relationships in the network itself.

The first method requires that applications be able to retrieve or at least parse Web content, the second requires interoperating with arbitrary communications protocols and the third requires some sharing of personal information with other users.

b) Identity Verification: Another feature which privacy-preserving social network should provide – and which current social networks do not – is a means for users to verify the online identities of their real-world friends. Today's social networks are already used to impersonate important politicians for comedic purposes [8], but the potential to use social networks for highly targeted phishing attacks is tremendous. Users should therefore be provided with an easy-to-use yet cryptographically strong means of verifying identity using limited out-of-band signalling (in person, over the phone, etc.).

A. Messaging

Users must be able to send each other messages, either in real-time (*a la* IM or Twitter) or a delay-tolerant fashion (*a la* e-mail or Facebook messages). Applications should be able to send such messages and cause them to be signed in desired.

B. Storage

Some applications – e.g. photo sharing applications – will require long-term storage within a pseudo-filesystem. Posting personal content in existing models is as straightforward as putting the content on the server and not restricting other users' access to it. In a client-centric system, however, access control must be done by client software.

Our system should provide applications with a general-purpose filesystem, and applications could share content, where permitted, by passing capabilities via the kernel.

C. Joint Content

One of the most interesting cases in the debate over online privacy is the case of *joint content*: content that was created and posted by one person, but which involves another person's name, likeness, opinions or comments. For instance, if a user is "tagged" in a Facebook photograph, they have the right to remove the tag – which recognises the stake that they hold in the content – but tagging another user's photo also requires the permission of the person who posted the photo.

We must provide the ability for applications to generate and share such content, and the joint content must be carefully tied by the kernel to its original context.

III. Privacy Mechanisms

Our application API must provide privileged operations that applications can execute, but its primary purpose is to safeguard the privacy of users' personal information. The

API, then, must hide personal information from applications unless it is truly required and user-authorised. The API should provide applications with the means to operate successfully without private information and, should they truly need it, privileged but limited operations which they can perform.

A. Access Control

Access control in existing social APIs is extraordinarily simplistic and permissive. In the case of Facebook, applications have some access to the personal information of almost all users, even those those who do use the application in question [9]. Access to other users' personal information depends on whether or not users have declared themselves to be "friends", which we have argued is an incorrect approach for a system that is concerned with privacy [10]. We propose that a privacy-preserving social network should not share information because users are in a list of friends, but that access control should be inferred from the normal course of user action [11], and that if friend lists exist, they should be derived from sharing policy, not the other way around.

In some cases, privacy policy should be simple: many applications will need either personal information or external interaction, but not both. Other applications, however, will have more sophisticated requirements, and our system must be capable of servicing their requests.

Application behaviour will be governed by a kernel-maintained policy. That policy should be restrictive by default but make it easy for capabilities to be granted by the course of normal user actions [11], i.e. without "security prompts" that distract users from their purposes for the network [12].

B. Placeholders

In many instances, applications do not actually require social information, it is placing them in a social context that adds value. Application code could use placeholders and pseudonyms in user interaction which the kernel could replace with actual social information, as suggested by Felt and Evans' *privacy-by-proxy* concept [13].

For instance, a chess application does not need to know my name or my opponent's, but it could generate messages such as "${opponent_name} has offered to resign" or tell the UI to "place the opponent's profile picture in the rectangle [50,0,100,100]" without seeing bitmap data.

C. Privileges

In our implementation, applications will run as plugins to a Java-based framework with no permission to perform system operations such as open network sockets. All external or inter-application interaction, then, must be accomplished via privileged operations provided by our API.

The atomic operations which it provides should be performed in such a way that their combination does not introduce unforeseen vulnerabilities.

Java's security policy does not allow us to restrict access to the current system time; if it did, we might even shut down covert schannels of information flow among malicious applications [14]. However, changing the flow of private information from unrestricted to covert-channel-only would be a significant improvement over current practice.

IV. Related Work

While Felt and Evans' concept of placeholders for social information [13] could be very useful, the utility of anonymized social networks is suspect given the ease with which these networks can be de-anonymised [15].

May, Gunter and Lee have previously proposed *Privacy APIs* [16], by which they mean the formalisation and analysis of legal policies in which access control requirements are supplemented by notification and logging requirements.

V. Conclusion

By recognising social application APIs as security APIs, we could provide users with a much safer social networking experience, giving them control over and visibility of what applications do with their personal information. We welcome the security API community's feedback on and involvement with these efforts.

References

[1] J. Bonneau, J. Anderson, R. Anderson, and F. Stajano, "Eight Friends Are Enough: Social Graph Approximation via Public Listings," in *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems (SNS '09)*, 2009.

[2] J. C. Perez, "Facebook's Beacon More Intrusive Than Previously Thought." http://www.pcworld.com/printable/article/id,140182/printable.html, Nov 2007. PCWorld.

[3] B. Stone, "Facebook Aims to Extend Its Reach Across the Web," *The New York Times*, vol. 12, no. 1, 2008.

[4] T. S. Schmidt, "Inside the Backlash Against Facebook." http://www.time.com/time/nation/article/0,8599,1532225,00.html, 2006. Time Magazine.

[5] E. Mills, "Facebook suspends app that permitted peephole." http://news.cnet.com/8301-10784_3-9977762-7.html, 2008. CNET News.

[6] J. Anderson, C. Diaz, J. Bonneau, and F. Stajano, "Privacy Preserving Social Networking Using Untrusted Servers," in *Proceedings of the Second ACM SIGCOMM Workshop on Online Social Networks (WOSN '09)*, 2009.

[7] D. M. Boyd, *Taken Out of Context – American Teen Sociality in Networked Publics*. PhD thesis, University of California, Berkeley, 2008.

[8] B. McGonigle, "Some profiles on MySpace.com not what they seem." http://www.boston.com/news/nation/washington/articles/2006/10/16/some_profiles_on_myspacecom_not_what_they_seem/, 2006. The Boston Globe.

[9] J. Bonneau, J. Anderson, and G. Danezis, "Prying Data Out of a Social Network," in *Proceedings of the 2009 International Conference on Advances in Social Network Analysis and Mining*, 2009.

[10] J. Anderson and F. Stajano, "Not That Kind of Friend: Misleading Divergences Between Online Social Networks and Real-World Social Protocols," in *Proceedings of the Seventeenth International Workshop on Security Protocols (SPW '09)*, 2009.

[11] K.-P. Yee, "Aligning security and usability," *IEEE Security and Privacy Magazine*, vol. 2, no. 5, pp. 48 – 55, 2004.

[12] A. Whitten, *Making Security Usable*. PhD thesis, Carnegie Mellon University, 2004.

[13] A. Felt and D. Evans, "Privacy Protection for Social Networking Platforms," in *Proceedings of Web 2.0 Security and Privacy 2008*, 2008.

[14] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613 – 615, 1973.

[15] L. Backstrom, C. Dwork, and J. Kleinberg, "Wherefore art thou r3579x?: anonymized social networks, hidden patterns and structural steganography," in *Proceedings of the 16th International Conference on the World Wide Web (WWW '07)*, pp. 181–190, 2007.

[16] M. J. May, C. A. Gunter, and I. Lee, "Privacy APIs: Access Control Techniques to Analyze and Verify Legal Privacy Policies," *Computer Security Foundations Workshop*, 2006.

The Usual Suspects

Mike Bond

Cryptomathic Ltd
327 Cambridge Science Park
Cambridge, CB4 0WG, UK
Mike.Bond@cryptomathic.com

Abstract. This paper outlines the key challenges facing security API design, emphasising the recurring problems which still have no practical solution – the usual suspects. The usual suspects are a starting point for designers making new contributions, and for verifiers looking for vulnerabilities. We discuss the problems of secure data structure building, data migration, secure cluster operation, and access control, drawing examples of failures from banking and from the Trusted Platform Module (TPM) API. Comprehensive treatment of these problems would be invaluable to industry API designers, who rarely have the resources to solve these problems afresh on any individual project.

1 Securely Building Data Structures

A regular security API challenge is to take small amounts of sensitive data which where each data item is individually encrypted and combine them together into a larger data structure which contains a lot of less sensitive data, but which is to be encrypted in its entirety under a transport or communications key. This problem comes up all the time. For example, in the smartcard industry one needs to build data structures to:

1. Embed cryptographic keys for EMV cards into a personalisation load data file to be sent to a physical personalisation machine (that loads all the data onto the smartcard).
2. Insert sensitive data into individual smartcard APDUs which of course will not be entirely encrypted (examples include personalisation of "native OS" cards, and EMV script processing)
3. Build database records consisting of a mixture of fields
4. Support creation of protocol messages with a key where there are a wide variety of protocol message formats or variations

There are several things that go wrong. API designers often try to build a *monotonic* API which stores intermediate state whilst the data structure is being built in encrypted blobs returned from one command, which are to be passed to the next. This allows the attacker to make many related ciphertexts and despite a potentially secure mode of operation, pull off a codebook attack by matching known and partially unknown ciphertext blocks.

2 Data Migration

Data migration is the challenge of getting cryptographic keys from one secure device to another. There are two factos to this: first, ensuring that the correct device receives the keys, and that this device will continue to enforce policy on behalf of the stakeholders, and second, ensuring that essential metadata associated with the keys is not lost or corruptible in the process.

We have seen dozens of problems with this, in banking HSMs, in the ubiquitous PKCS#11-based HSMs that are used in Certificate Authorities and much more, and in many other areas too. The author draws on an example from the Trusted Computing Group TPM version 1.2, where a data typing failure in capability tokens almost causes a security breach.

3 Secure Clustering

Secure clustering is the challenge of assembling together devices implementing the same API and policy into a cohesively functioning unit. The basic functionality is dead simple: store keys externally under a master key, and load the same master key into each device. Things start to get more complicated when you consider access control problems, and how to enforce more advanced policies. For instance, if you want to ensure a crypto key cannot be used more than 1000 times a day, how can you have a functioning cluster which is resilient to failure and able to recover, yet which still keeps track of the usages made, and does not simply allow each node to make 1000 uses in its own right?

There have been solutions to this recurring problem, but many of them try to bind together the cluster into an integrity preserving state where any deviation or problems will cause the system to lock up and stop working. Administrators must attend under dual control and examine all devices and software (a heavy burden) before the system can be unlocked. Such solutions are fragile. Yes, they may work, but they fall prey to the "just click Yes" security usability issues that plague web browsers to this day when considering SSL.

4 Access Control

Sadly the usability of access control mechanisms that govern access to sensitive security API commands is in a pretty poor state. They range from sophisticated Role-Based Access Control (RBAC) systems as offered by IBM's Common Cryptographic Architecture, through the one-bit "physical presence" switch for the TPM, to a variety of smartcard and secret sharing mechanisms used by many cryptographic devices. Across the board these mechanisms always seem to fall short: a typical case is where the calling application is left with a password security problem, having to store and protect a password whose discovery totally compromises the entire system, yet is needed on a regular basis to get work done.

Access control mechanisms are often overlooked in analysis of APIs (and indeed protocolss), maybe because the analysts don't have a clear model of how

the access control might be configured. The author aims to flesh out the typical access control environment surrounding a security API, and to point to some challenges where an analytical method which can cope with a major temporal component could be useful.

APIs In The Real World – a users perspective

(How APIs are used and abused)

George French

Barclays Bank PLC.

1234 Pavilion Drive,

Northampton, NN4 7SG, UK

George.French@Barclays.com

Abstract. This paper describes why organizations like banks use cryptography, what they expect from it and how they currently attempt to achieve this through the use of cryptographic APIs. It will describe the problems faced in selecting and using an API. It will provide examples of what goes wrong. It will then proceed in describing what the "perfect API" might look like from a use and deployment perspective.

1. Why Banks use cryptography

The use of cryptography by banks has been driven by a number of different stimuli over an extended period of time. This has led to an organic growth in the use of cryptography.

Very early on the main use of cryptography was for data integrity, to prevent/detect the malicious or accidental alteration of data. With the adoption of networking within the organization encryption was deployed to provide a level of privacy.

The next driver for the use of cryptography was as a control mechanism in order to manage risk. This not only includes fraud but business and operational risk as well.

Latterly the drivers are compliance and regulation from government and organizations and heightened public awareness of Identity Theft and data breaches e.g.

1. VISA, Mastercard, SWIFT, PCI etc. for scheme compliance
2. Data Protection Act UK, BDSG in Germany and EU Data Protection Directive
3. And as an indirect result of other Legislation e.g. SARBOX, GLB Act
4. TJ Max and HMRC Data Loss.

This has resulted in cryptography being viewed as a control to mitigate or manage risk or as a tool to demonstrate compliance, and not just as a means of providing confidentiality, integrity or authenticity. By being use in this way cryptography and the APIs that support it are also subject to additional requirements. The paper will outline these requirements and there relevance.

2. How Banks select an API to use.

The author will outline what the drivers are for the selection of an API and provide insight into how cryptography is typically deployed :

- Traditional banking functions verses general purpose functions
- Bespoke cryptography mechanisms
- The location and infrastructure of the application requiring cryptography
- Who will be undertaking the development or intergeneration of the API into the application or will a product be purchased.
- Current deployed solutions
- Governance (internal and external)
- Cost

3. Common pit falls found.

The author will provide examples of the most common errors and omissions encountered when designing application to use cryptographic APIs as well as a few examples of not so common errors.

This will include the issues with developing in-house as well as the review of vendors use of cryptographic APIs.

3. What the perfect API might look like

The author will describe from the implementation and deployment perspective the attributes of the "perfect" API. This will include a layering approach which has proved successful in meeting the additional requirements a cryptographic API has to fulfill within the banking sphere, as well as preventing implementation errors which lead to security weakness in applications.

Improving PIN Processing API Security

R. Focardi and F. Luccio
Università di Venezia, Italy
focardi@dsi.unive.it

G. Steel
LSV, CNRS & ENS de Cachan, France
graham.steel@lsv.ens-cachan.fr

Abstract

We propose a countermeasure for a class of known attacks on the PIN processing API used in the ATM (cash machine) network. This API controls access to the tamper-resistant Hardware Security Modules where PIN encryption, decryption and verification takes place. The attacks are differential attacks, whereby an attacker gains information about the plaintext values of encrypted customer PINs by making changes to the non-confidential inputs to a command. Our proposed fix adds an integrity check to the parameters passed to the command. It is novel in that involves very little change to the existing ATM network infrastructure.

1 Introduction

In the international ATM (cash machine) network, users' personal identification numbers (PINs) have to be sent encrypted from the PIN entry device (PED) on the terminal to the issuing bank for checking. Issuing banks cannot expect to securely share secret keys with every cash machine, and so the PIN is encrypted under various different keys as it passes through the network. Typically, it will first be encrypted in the PED under a key shared with the server or *switch* to which the ATM is connected. The PIN is then decrypted and re-encrypted under the key for an adjacent switch, to which it is forwarded. Eventually, the PIN reaches a switch adjacent to the issuing bank, by which time it may have been decrypted and re-encrypted several times. The issuing bank has no direct control over what happens in the intermediate switches, so to establish trust, the internationally agreed standards ANSI X9.8 and ISO 9564 stipulate the use of tamper proof cryptographic hardware security modules (HSMs). In the switches, these HSMs protect the PIN encryption keys, while in the issuing banks, they also protect the PIN derivation keys (PDKs) used to derive the customer's PIN from non-secret validation data such as their personal account number (PAN). All encryption, decryption and checking of PINs is carried out inside the HSMs. To this aim, the HSMs have a carefully designed API providing functions for *translation* (i.e., decryption under one key and encryption under another) and *verification* (i.e. PIN correctness checking). The API has to be designed so that even if an attacker obtains access to the host machine connected to the HSM, he cannot abuse the API to obtain customer PINs.

In the last few years, several attacks have been published on the APIs in use in these systems [5, 6, 8]. Very few of these attacks directly reveal the PIN. Instead, they involve the attacker calling the API commands repeatedly with slightly different parameter values, and using the results (which may be error codes) to deduce the value of the PIN. High-profile instances of many PINs being stolen from a hacked switch has increased interest in the problem [1]. Recently, a Verizon Data Breach report and a subsequent article in the press confirmed publicly for the first time that PINs are being extracted from HSMs on a wide scale [3, 2].

PIN recovery attacks have been formally analysed before, but previously the approach was to take a particular API configuration and measure its vulnerability to combinations of known attacks [10]. In recent work, we proposed an extension to language based information flow analysis to take account of cryptographic primitives designed to assure data integrity, in particular MACs [7]. We showed how PIN processing APIs could be extended with MACs to counteract differential attacks, and showed how this revised API type-checked under our framework. However, that work was rather theoretical, and did not attempt to explain how our proposal could be put into practice. In particular, for our proposal to be

feasible in the short term, it needs to be adapted to take into account the constraints of the existing ATM infrastructure. In this paper we outline what we believe to be a practical scheme. We assess its impact on security and the amount of changes that would be required to the ATM network to put it in to practice.

We will not review the operation of PIN processing APIs in detail here. For understanding the abstract, it suffices to know that the attacks are caused by an attacker making illegitimate queries to the API, 'tweaking' the bits of the non-confidential parameters such as the customer's PAN and the *decimalisation table* (dectab). Interested readers are referred to existing literature for more details [5, 6, 8, 10].

2 The Fix

In another paper [7], we show how differential attacks can be countered by the use of MACs, which prevent the intruder from making arbitrary queries to the verification and translation APIs. Only queries whose parameters match the supplied MAC are processed. However, the infrastructure changes needed to add full MAC calculation to ATMs and switches are seen as prohibitive by banks [4]. We propose here a way to implement a weaker version of our scheme whilst minimising changes to the existing infrastructure. We lose some security, since our MACs now have an entropy of only 5 decimal digits ($2^{16} < 10^5 < 2^{17}$). We assess the effect of this change in section 2.4.

2.1 Ideal MAC-based integrity

The idea proposed in our paper [7] is to add a MAC of the non-confidential parameters required for PIN verification to the input to the PIN verification command. The PIN_Verify command checks the MAC before performing the PIN calculation. If the MAC check fails, the command halts without checking the PIN. This way we achieve 'robust declassification' [9], i.e. we declassify only the result of the legitimate comparison of the encrypted PIN block with the real PIN, and nothing else.

In our paper, we did not discuss how exactly this MAC should be calculated, and where it should be stored. Below we propose a way to store a MAC of the non-confidential inputs to the verification command on the card itself, using an existing mechanism.

2.2 CVC/CVV Codes

We observe that cards used in the cash machine network already have an integrity checking value: the card verification code (CVC) or value (CVV) is a 5 (decimal) digit number included on the magnetic stripe of most cards in use. It is, in effect, a MAC of the customer's PAN, expiry date of the card and some other data. The current purpose of the CVV is to make copying cards more difficult, since the CVV is not printed on the card and does not appear on printed POS receipts[1]. Below we give the algorithm for calculating the CVV. This is done at the card issuing facility. CVVs are checked at the verification facility.

PAN	Exp date	Service code	0 pad
16 digits max	4 digits	3 digits	9 digits max
Block B1	Block B2		

Note the partition of the CVV plaintext field into two blocks, B1 and B2. To construct the CVV, a two-part DES key is required. Call the two 64-bit parts key K1 and key K2. The hexadecimal CVV root is constructed as

[1]The CVV/CVC is not to be confused with the CVC2 or CVV2, which is printed on the back of the card and designed to counteract 'customer not present' fraud when paying over the Internet or by phone.

$$CVV_{hex} = enc(K1, dec(K2, enc(K1(enc(K1,B1) \oplus B2))))$$

The 5 digit decimal is constructed using the Visa decimalisation scheme:

1. Extract all decimal digits from CVV_{hex}, preserving their order from left to right.
2. Left justify the result
3. Reduce any remaining digits by 10
4. Left justify the result and append it to the result of 2
5. The CVV is the first 5 digits (from left to right) of the result.

2.3 Packing the MAC into the CVC/CVV

Our proposal is to pack more information into the CVV at issue time, and to use this as a MAC at the verification facility. In our formal scheme, we included the data of the decimalisation table and card offset. Observe that with the maximum 16 digits of the PAN being used, we still have 9 digits of zeros in the final field of block B2. Our idea is to use the CVV calculation method twice, in the manner of a hashed MAC or HMAC function. We will calculate the CVV of a new set of data, containing the decimalisation table and offset or PVV and a code for the PIN block format. Then we will insert the result of the original CVV calculation to produce a final 5-digit MAC.

Our second CVV, which we will call CVV', contains the following fields:

Dectab	Offset/PVV	PIN block format	original CVV	0 pad
16 digits max	4 digits	1 digit	5 digits	6 digits max
Block B1'	Block B2'			

We calculate CVV' in the same way as the standard CVV. This makes for easy upgrade from the original infrastructure, because CVV generation and verification commands are already available in HSMs so will need minimal changes to the firmware. The PIN_Verify command of the HSM must be changed to check CVV' before performing a verification test. The PIN test is only performed if the CVV' of the inputs matches the supplied CVV'. Of course, the API of the HSM must not make available the functionality to allow the creation of CVV's on arbitrary data.

The scheme is practical because ATMs and switches generally already send the CVV from the ATM to the issuing bank, so can easily be adapted to send CVV'. In fact, many ATMs blindly send all the 'Track 2 data' from the magnetic card - this includes the PAN, expiry date, and CVV. Under most schemes there is still space on the magnetic stripe for a further 5 digit code. Chip based cards should have no problem storing a further 20 bits of data. So, we could use CVV' and the original CVV', thus allowing CVVs to be checked separately if required.

One could use the same keys for calculating the CVV' as for the CVV, or one could use different keys. Either way, the verifying HSM needs access to these keys. The use of different keys could be motivated by a desire to be able to check CVVs, and so to an extent to verify card authenticity, in ATMs, without giving them the keys used to create CVV's.

2.4 Security of the CVV Based Scheme

In our formal scheme we assume a perfect, collision free scheme. However, in proposing a scheme with a 5-digit MAC value, we are admitting the possibility of brute-force attacks. To guess the CVV' for a given set of parameters should take an average of 50 000 trials. So, an attack like the dectab attack which previously required 15 calls to the API will now take 750 000 calls. HSMs typically perform something of the order of 1000 PIN verifications per second, so this change moves the expected attack time for a single PIN from 0.015 seconds to 750 seconds, or 12.5 minutes, making the 'lunch hour hack' scenario of [6] worth an expected 4 or 5 PAN/PIN pairs.

3 Conclusions

We have described a version of our MAC based scheme for ensuring integrity of queries to PIN processing APIs that is easy to implement and does not require wholesale changes to the ATM infrastructure. This is at the cost of some security, since the CVV codes can be cracked by brute force, but its implementation in the short term would make attack scenarios far less profitable. In the medium term we feel that the full MAC scheme should be used. The cost of this should be weighed against the cost of a complete overhaul of the way PIN processing is carried out in the ATM network.

References

[1] Hackers crack cash machine PIN codes to steal millions. The Times online. http://www.timesonline.co.uk/tol/money/consumer_affairs/article4259009.ece.

[2] PIN Crackers Nab Holy Grail of Bank Card Security. Wired Magazine Blog 'Threat Level'. http://blog.wired.com/27bstroke6/2009/04/pins.html.

[3] Verizon Data Breach Investigations Report 2009. Available at http://www.verizonbusiness.com/resources/security/reports/2009_databreach_rp.pdf.

[4] R. Anderson. What we can learn from api security. In *Security Protocols*, pages 288–300. Springer, 2003.

[5] O. Berkman and O. M. Ostrovsky. The unbearable lightness of PIN cracking. In Springer LNCS vol.4886/2008, editor, *11th International Conference, Financial Cryptography and Data Security (FC 2007), Scarborough, Trinidad and Tobago*, pages 224–238, February 12-16 2007.

[6] M. Bond and P. Zielinski. Decimalization table attacks for pin cracking, 2003. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-560.pdf.

[7] M. Centenaro, R. Focardi, F. Luccio, and G. Steel. Type-based analysis of PIN processing APIs. Available from http://www.dsi.unive.it/~focardi/typing-PIN-full.pdf, 2009.

[8] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[9] A.C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, May 2006.

[10] G. Steel. Formal Analysis of PIN Block Attacks. *Theoretical Computer Science*, 367(1-2):257–270, November 2006.

Compiling Applications with Information-Flow Policies to Systems with Trusted Modules (Work in Progress)

Jérémy Planul[1] and Cédric Fournet[2,1]

[1] MSR-INRIA
[2] Microsoft Research

Today, many computers come bundled with some form of secure coprocessor with a dedicated secure instruction set—for example, most laptops now embed a Trusted Platform Module (TPM) [10], and many high-end processors feature a special `skinit` instruction [1]. These instructions can run a given piece of code in isolation, with strong code-based identity and privileged cryptographic operations, for instance to seal some persistent state or to perform remote attestation. Such hardware mechanisms can greatly reduce the TCB, removing the need to trust most of the code for the host operating system and application, and thus help protect critical data and computations from malicious software. For example, TPMs are commonly used for secure booting, e.g. Bitlocker [7] guards access to the master keys for disk encryption, so that the disk content is released only after authenticating the operating systems. Research papers also describe e.g. how to build secure online payment systems [2], and how to use `skinit` to run small pieces of application code in isolation [6]. Still, the secure instructions are remarkably seldom used in practice. We believe that the complexity of their low-level interface and the lack of programming tools are major obstacles to their mainstream adoption for writing security applications.

At a more abstract level, language-based security often relies on information flow policies [8]. Each variable is given a level in a security lattice; this level indicates the intended integrity and confidentiality of any information stored in this variable. Thus, a program is deemed secure (non-interferent) if an adversary that can access only low-level information cannot gain (or influence) any higher-level information by executing the program. Static analyses and type systems have been developed to verify that a program is secure with regards to a given policy. Further, it is sometimes possible to compile such programs to a given system while preserving all their security properties; hence, Jif and FlowCaml [9] provide security typechecking for Java and Caml, respectively. Further, Jif/Split [12,13] and Swift [3] can automatically partition distributed programs into local code, each running at a given security level, representing for instance the level of trust granted to every host in a protocol, subject to some locality constraints. This compilation method is partial: in some cases, the security policy may be such that no host has a sufficient level to run some parts of the computation (for instance when combining secret information from mutually-suspicious parties.). Cryptographically-blinded evaluation techniques [11] can solve this problem in some cases, but with a high performance penalty. Instead, it is tempting to rely

on secure hardware capabilities to virtually “boot” short-lived, highly-trusted execution environments at any host that supports them. For example, a server may provide code that locally interacts with a client, and both the client and the server may trusts (some of) their secrets to secure evaluation at the client. To this end,

- We define an operational semantic for a core subset of the secure instructions, by extending the semantics of a core imperative programming language. Our semantics aims at formal simplicity while still reflecting the main security features of the hardware specifications at a level of details sufficient for reasoning about information flows.
- We develop a prototype compiler for a typed information flow language that partitions code and automatically takes advantage of the secure instructions.
- We show that our compilation scheme preserves information-flow security, at least for our target semantics, under standard assumptions on cryptographic primitives.

We supplement a simple imperative language with data representations for code, and with instructions to (1) run some code in isolation; (2) compute a secret associated with the code being executed; and (3) increment monotonic counters. We also consider encodings of additional security functionality on top of these instructions, for instance to (4) cryptographically seal and unseal data using a key associated with some code; (5) cryptographically sign and verify data using a key associated with some code and attested by a TPM certificate.

We use this language as the target of a new security-preserving compiler, built by adapting recent work on cryptographic support for information-flow policy enforcement [5,4]. In their work, imperative commands can be annotated with locality information that indicate where a sub-command should be executed. Each locality is also given a security level, used both for typechecking the source program, and to generate cryptographic operations to secure the implementation. Their compilation process involves the generation of a protocol for securing the transfer of control between locations, as specified by the control flow of the source program, and selective encryption and authentication for securing the exchange of data. In addition, we provide runtime support for implementing highly-trusted localities by translation to our (formal) secure instruction set. Hence, we obtain distributed code, composed of ordinary application code plus high-security code, together with custom cryptographic support to coordinate their execution, so that the information-flow properties of the source program still hold in the resulting system.

References

1. Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD publication no. 33047 rev 3.01, May 2005.
2. S. Balfe and K. G. Paterson. e-EMV: Emulating EMV for internet payments using trusted computing technology. *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing (STC 2008)*, pages 81–92, 2008.

3. S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Communications of the ACM*, 52(2):79–87, 2009.
4. C. Fournet, G. le Guernic, and T. Rezk. From information-flow policies to cryptographic mechanisms: a security-preserving compiler for distributed programs, 2009. Draft, at http://www.msr-inria.inria.fr/projects/sec/cflow.
5. C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *35th Symposium on Principles of Programming Languages (POPL'08)*, pages 323–335. ACM Press, Jan. 2008.
6. J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Apr. 2008.
7. Microsoft. Windows BitLocker drive encryption, 2006. http://technet.microsoft.com/en-us/library/cc766200.aspx.
8. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.
9. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003. ©ACM.
10. Trusted Computing Group. PC client specific TPM interface specification (TIS). version 1.2, revision 1.00, July 2005.
11. A. Yao. Protocols for secure computation. In *Twenty-third IEEE Symposium on Foundations of Computer Science*, pages 160–164, Nov. 1982.
12. S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, 2002.
13. L. Zheng, S. Chong, A. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *15th IEEE Symposium on Security and Privacy*, 2003.

Extraction of properties in C implementations of security APIs for the verification of Java applications

Cyrille Artho* Yutaka Oiwa* Kuniyasu Suzaki* Masami Hagiya†

Abstract

Java applications utilize various security APIs for cryptography and access control, such as available through packages java.security and javax.crypto. For performance reasons, these libraries internally use an implementation written in C, accessed through the Java Native Interface. Our goal is to extract properties in the source code of the C library, and translate these assertions back into the Java domain. This allows these properties to be used in verification of Java code, opening up various applications that are not possible when verifying binary code.

Keywords: Software model checking, software testing, Java, C, API, verification, security

1 Introduction

Java offers various security and cryptography facilities [11]. Security includes permission management (access control managed by security policies) and secure class loading. Cryptography includes access to cryptographic functions, such as hashing and encryption ciphers, and cryptographic protocols [16].

Java applications request security services from the Java platform. In order to accommodate for differences between different platforms, implementations are typically not provided by the standard library itself. Instead, they are accessed via *providers,* which encapsulate algorithm-specific implementations behind a standardized application programming interface (API). This allows an application to be independent of a provider. The same kind of function can be implemented by different providers, and exchanged if necessary [16].

For instance, certain access control features may only be available if the underlying operating system or file system supports corresponding mechanisms. Secure class loading may take advantage of specific hardware (trusted computing) [7, 21]. Cryptographic functions implemented in hardware or software may have to be replaced by different implementations if an existing implementation is too slow [1], or if an underlying cryptographic algorithm has been broken, i. e., demonstrated not to be cryptographically secure anymore [9].

Because the underlying functionality and API are very system-specific and low-level, the software library is usually implemented in C. Table 1 shows how in a Java application, access to C code is provided by the Java Native Interface (JNI) [11, 17]. This causes a problem when a Java application is analyzed by Java-specific tools, such as software model checkers [22]. An analysis tool may not be able to inspect or execute C code. In model checking, the inability to handle actions outside the given bytecode platform is a well-known problem when analyzing application that use library functionality such as network communication [5]. Because the effect of C code cannot be controlled by the Java platform, analysis tools may provide incorrect results. For model checking, a tool has to be able to restore the entire program state to a previous state. If C code is involved, side-effects of its execution may prove to be impossible to revert. This effectively puts programs using JNI calls outside the range of model checkers.

We plan to extract properties from low-level C code that are relevant for the correct behavior of Java applications. In this way, we can apply various inspection and analysis techniques to Java programs that are not possible otherwise.

Compared to similar work [19, 18], we plan to generate executable code instead of property annotations. We think that existing toolkits for code analysis that represent the abstract syntax tree as XML data may be the appropriate platform for a unified representation of the data [10, 14]. Mapping rules can then relate C code fragments to Java code.

*Research Center for Information Security (RCIS), AIST, Tokyo, Japan

†University of Tokyo, Tokyo, Japan

Table 1: Architecture of Java application using Java library backed by native code.

Layer	Language	Description
Application	Java	Written by developers, target of verification in this project
Java library	Java	High-level functions (e. g., security and cryptography)
JNI layer	Java	Java Native Interface: passes library calls to low-level code
JNI impl.	C	C counterpart of JNI, sometimes automatically generated
Crypto library	C	Library implementing low-level functions
Device driver	C	(If present) interface to hardware (e. g., trusted computing)

2 Benefits

There are several benefits when low-level C code is modeled in the same language as the target application:

- Better integration into the analysis tool, as the tool can fully inspect properties of interest.
- The possibility of combining properties of multiple implementations, giving a stronger specification for verification.
- The possibility of using other analysis technologies, such as symbolic execution, model checking, or fault injection. These technologies are usually not applicable to low-level code.

Model checking for software is specifically useful for concurrent applications, as the outcome of all possible thread and communication schedules cannot be tested effectively. A test run covers one particular scenario [15]. In software where multiple threads [20] of execution work in parallel, a test run executes one particular thread schedule. As the schedule is typically non-deterministic, even repeated test runs cover only a part of all behaviors. Different verification approaches are required for more exhaustive verification. Model checking has the advantage that it is fully automated, but given verification tools for Java require that the entire application exists as Java bytecode [22] or that side-effects of system-specific code are modeled by a special library [4].

Similarly, fault injection tools also require that code is available in a platform that the tool supports [3, 2]. Conversion of so-called checked exceptions from JNI to Java would allow such tools to have a richer view of the library, including exceptions returned from C code.

As complex computations are inevitably simplified when extracting only key properties, the resulting model code would also be more efficient than the original one. This is another benefit both for model checking and other analysis types, because analysis can scale to larger applications.

3 Implementation Strategy

A model of a library function may consist of a *stub,* implementing only a subset of the real functionality [6]. The stub has to be precise enough to allow for execution of a test case of interest. For cryptographic functions and security APIs, certain properties of their behavior help us to write such stubs:

- Cryptographic functions can be replaced with a stub that either returns clear text (for matching keys) or a pseudo cipher text that differs in a simple way. For example, each string may be preceded with a special marker character to mark it as encrypted. This marker is removed upon encryption. Because the goal of software verification is only to ensure that encryption is used whenever necessary, the lack of security of this "encryption scheme" is not a problem.

Java interface:

```
public final static native void
TPM_NONCE_nonce_set(long jarg1, TPM_NONCE jarg1_, short[] jarg2);
```

C implementation:

```
SWIGEXPORT void JNICALL
Java_iaik_tc_tss_impl_jni_tsp_TspiWrapperJNI_TSS_1NONCE_1nonce_1set(
  JNIEnv *jenv, jclass jcls, jlong jarg1, jobject jarg1_, jshortArray jarg2) {
    // other declarations omitted
    if (jarg2 && (*jenv)->GetArrayLength(jenv, jarg2) != TPM_SHA1BASED_NONCE_LEN) {
        SWIG_JavaThrowException(jenv, SWIG_JavaIndexOutOfBoundsException,
                                "incorrect array size");
        return;
    }
...
```

Figure 1: C implementation of a Java native method.

- Security APIs often work in a binary way, either granting or denying access. This can be modeled as a non-deterministic decision.

In both cases, the exact way the C security library works is often irrelevant for testing an application. The library has to implement high-level properties such as providing a secure one-way hash function. Such properties can be analyzed in isolation of the application, for example through cryptanalysis. When analyzing the Java application, only correct usage of the functionality is important.

Therefore, stubs should model preconditions that the Java application must meet when calling the API. Such preconditions can be extracted from assertions in the C implementation. Other properties, such as a correct sequence of calls, may also be accessed by more advanced inspection techniques on the C code, such as program slicing [13].

Figure 1 shows a part of the API for Trusted Computing for Java [12, 21]. In this code, method `nonce_set` is declared to be native in Java, and implemented in C. The Java Native Interface declaration requires the expanded class name of the method and a lengthy signature, but the interesting part is the C implementation of the method. In the C code, the array length of the last argument is checked against a constant that is defined elsewhere. This check is not part of the Java program! However, knowledge of JNI calling conventions allows for a translation of the if-expression from C to Java, where it can be verified even if the C code is subsumed by a stub.

Previous work has implemented a similar mapping for the verification of low-level C libraries [19, 18]. The focus was on generating code annotations, but we aim at generating executable code that does not require extra tool support for analysis. By leveraging tools that represent program structure in XML form, we have a unified representation of the problem [10, 14]. Finally, we hope to include recent advances in reverse engineering to infer properties relating to correct sequences of API calls [8].

References

[1] T. Arnold and L. Van Doom. The IBM PCIXCC: a new cryptographic coprocessor for the IBM eServer. *IBM J. Res. Dev.*, 48(3–4):475–487, 2004.

[2] C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. Int. Conference on Formal Methods (FM 2006)*, Canada, 2006.

[3] C. Artho, A. Biere, and S. Honiden. Exhaustive testing of exception handlers with enforcer. *Post-proceedings of 5th Int. Symposium on Formal Methods for Components and Objects (FMCO 2006)*, 4709:26–46, 2006.

[4] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Efficient model checking of networked applications. In *Proc. TOOLS EUROPE 2008*, volume 19 of *LNBIP*, pages 22–40, Zurich, Switzerland, 2008. Springer.

[5] C. Artho, W. Leungwattanakit, M. Hagiya, and Y. Tanabe. Tools and techniques for model checking networked programs. In *Proc. SNPD 2008*, Phuket, Thailand, 2008. IEEE.

[6] E. Barlas and T. Bultan. Netstub: a framework for verification of distributed Java applications. In *Proc. 22nd Int. Conf. on Automated Software Engineering (ASE 2007)*, pages 24–33, Atlanta, USA, 2007. ACM.

[7] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A practical guide to trusted computing*. IBM Press, 2007.

[8] Vitaly Chipounov and George Candea. Reverse-Engineering Drivers for Safety and Portability. In *4th Workshop on Hot Topics in System Dependability (HotDep)*, San Diego, USA, 2008.

[9] H. Dobbertin. Cryptanalysis of MD4. *J. Cryptology*, 11(4):253–271, 1998.

[10] K. Gondow, T. Suzuki, and H. Kawashima. Binary-level lightweight data integration to develop program understan ding tools for embedded software in C. In *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC 20 04)*, pages 336–345, Washington, USA, 2004. IEEE Computer Society.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[12] Institute for Applied Information Processing and Communications. *Trusted Computing for the Java Platform*, 2009. `http://trustedjava.sourceforge.net/`.

[13] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[14] K. Maruyama and S. Yamamoto. A tool platform using an XML representation of source code informati on. *IEICE – Trans. Inf. Syst.*, E89-D(7):2214–2222, 2006.

[15] D. Peled. *Software Reliability Methods*. Springer, 2001.

[16] Sun Microsystems, Santa Clara, USA. *How to Implement a Provider in the Java Cryptography Architecture*, 2009. `http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/HowToImplAProvider.html`.

[17] Sun Microsystems, Santa Clara, USA. *Java Native Interface*, 2009. `http://java.sun.com/javase/6/docs/technotes/guides/jni/`.

[18] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Proc. 17th Conf. on Security Symposium (SS '08)*, pages 365–377, San Jose, USA, 2008. USENIX Association.

[19] G. Tan and G. Morrisett. ILEA: inter-language analysis across Java and C. In *Proc. 22nd annual ACM SIGPLAN Conf. on Object-oriented programming systems and applications (OOSPLA 2007)*, pages 39–56, Montreal, Canada, 2007. ACM.

[20] A. Tanenbaum. *Modern operating systems*. Prentice-Hall, 1992.

[21] T. Vejda, R. Toegl, M. Pirker, and T. Winkler. Towards trust services for language-based virtual machines for grid computing. In *Proc. 1st Intl. Conf. on Trusted Computing and Trust in Information Technologies (Trust '08)*, pages 48–59, Villach, Austria, 2008. Springer.

[22] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

Secure your PKCS#11 token against *API attacks!*[*]

M. Bortolozzo, G. Marchetto, R. Focardi
Università di Venezia, Italy
focardi@dsi.unive.it

G. Steel
LSV, CNRS & ENS de Cachan, France
graham.steel@lsv.ens-cachan.fr

Abstract

PKCS#11 defines a widely adopted API for cryptographic devices such as USB crypto-tokens and smartcards. Despite its widespread adoption, PKCS#11 is known to be vulnerable to various attacks that enable the extraction (as plaintext) of sensitive keys stored on the device. These attacks have been formalized and analyzed via model-checking, so as to automatically find flaws on specific subsets of PKCS#11 and on given device configurations. The analyses, however, are performed on an abstract model of the standard and the 'theoretical' attacks have to be tried by hand on real devices. In this paper we shortly describe a new tool, named *API attacks!*, that aims at automatically performing the above mentioned analyses on real PKCS#11 devices. We believe this tool might be helpful both to hardware developers, willing to improve the security of their existing and new devices, and to system administrators that might want to check their device is configured is a secure way before distributing it to end-users.

1 Introduction

PKCS#11 defines a widely adopted API for cryptographic devices such as USB crypto-tokens and smartcards. As well as providing access to cryptographic functionality, the interface is suppose to preserve certain security properties, e.g. no matter what sequence of commands is called by the application, the values of keys stored on the device and marked as *sensitive* should never become known 'in the clear'. However, PKCS#11 is known to be vulnerable to various attacks that compromise this property.

PKCS#11 has been formalized and analyzed via model-checking, allowing the automatic detection of attacks on specific subsets of the API and on given device configurations [3]. The analyses, however, are performed on an abstract model of the standard and the 'theoretical' attacks have to be tried by hand on real devices. Particular PKCS#11 compatible devices may implement the standard in subtly different ways to try to prevent the attack. What is needed is a way to link the abstract model checking analysis to the API as implemented on real devices.

In this paper we describe a new tool, named *API attacks!* [1], that aims at automatically performing the above mentioned analyses on real PKCS#11 devices. In summary, the tool (i) attempts a set of known attacks reporting the results; (ii) reads the actual subset of PKCS#11 implemented on the token; (iii) generates a model of the specific subset of the standard and gives it as an input to the model checkers NuSMV and SATMC; (iv) parses the results of the model checkers and tries to mount the attacks on the real token;[1] (v) performs custom attacks and static checks on the actual token configuration, pointing out potential sources of flaws.

We believe this tool might be helpful to hardware developers, willing to improve the security of their existing and new devices. In order to circumvent the flaws on PKCS#11, hardware developers usually modify or patch the standard via proprietary extensions. *API attacks!* could be used to try all the existing known attacks on new extensions, reporting where the attack possibly fails, and, more interestingly, could incorporate a model of a new proposed extension so to analyse it via model checking looking for possible new flaws or variants of existing ones. It might be the case, in fact, that a new patch blocks all

[*]Work partially supported by Miur'07 Project SOFT: "*Security Oriented Formal Techniques*"

[1]This last feature is currently under development: at the present state the tool just reports the results of (iii) to the user.

the known attacks but can be circumvented by non trivial variations of the attack sequences. This could be detected automatically via model-checking, once the proprietary extension is modelled in the tool.

The tool might also be interesting for system administrators, who want to configure tokens in a secure way before distributing them to end-users. Attacks, in fact, are sometime based on 'weak' token configurations where, e.g., keys can be used with different, conflicting, roles (see section 2). We thus believe that having the possibility of automatically checking a specific configuration against known attacks (and variants of those, via model-checking) could be extremely valuable.

2 Attacks on PKCS#11

In this section, we illustrate some of the attacks checked by the tool. They are all sequences of legal API invocations that lead to the leakage of *sensitive* keys, i.e., keys intended to be securely stored in the token and never extracted, unless encrypted under other suitable keys called *key encryption keys*. This latter event is useful, e.g., when we need to share a new key between two devices that already share a key encryption key *kek*: the new key k is *wrapped* under *kek* obtaining a ciphertext $\{k\}_{kek}$. This wrapped key is exported from the first device and imported in the second one. Only when the ciphertext is inside the second device, the *unwrap* occurs: k is security stored and, from now on, it can be used for encrypted communication between the two devices.

This simple wrap/unwrap mechanism is often source of attacks, if we mix the role of the keys. The standard, in fact, does not forbid having keys with attributes that specify different uses like, e.g., *wrap* and *decrypt*. Unfortunately this leads to simple attack sequences as the following one. From now on, we write $\&k$ to denote the *handle* of key k stored in the device.

1. `wrap(`$\&k$`,`$\&k$`)` gives $\{k\}_k$
2. `decrypt(`$\{k\}_k$`,`$\&k$`)` gives k

This attack is a variant with just one key of the key separation attack presented in [6]. Intuitively, key k is wrapped under itself, via a call to `wrap`, and the obtained ciphertext is decrypted with k, by calling `decrypt`, obtaining k as plaintext. Notice that this decryption occurs in the token with no knowledge of k (only the handle $\&k$ is needed).

It seems thus important to forbid this double role on the same key. This can be done, e.g. by 'patching' the API so that attributes *wrap* and *decrypt* are sticky, i.e., once set they cannot be unset, and can never be set together. However a subtler variant of the attack might be performed as follows. Let k_e be a key generated by the attacker and k_u a key, stored in the device, that can be used both to *unwrap* and *encrypt*:

1. `encrypt(`k_e, $\&k_u$`)` gives $\{k_e\}_{k_u}$
2. `unwrap(`$\{k_e\}_{k_u}$, $\&k_u$`)` imports k_e in the device returning $\&k_e$
3. `set_wrap(`$\&k_e$`)` sets the *wrap* attribute for k_e
4. `wrap(`k, $\&k_e$`)` gives $\{k\}_{k_e}$
5. the intruder decrypts $\{k\}_{k_e}$ obtaining k

This attack has been discovered in [3], via model-checking. Intuitively, the intruder encrypts his key under k_u (1). This allows him to import the key via an `unwrap` call (2). Once the key is in the device, he sets the *wrap* attribute (3) and just wraps the sensitive key k with k_e (4). The intruder can now decrypt $\{k\}_{k_e}$ with k_e, which he knows, so this last event is performed outside the device.

Even subtler attacks can be mounted by, e.g., unwrapping twice the same key so to obtain two different instances of the very same key. This allows the intruder to set two conflicting attributes on the same key by just setting one attribute to each identical copy. This makes it even more difficult

to circumvent the above attacks since the conflicting attribute policy should extend to all the identical copies of the very same key. The interested reader is referred to [2, 6, 3] for more detail.

3 The *API attacks!* tool

The *API attacks!* tool aims to allow hardware developers and administrators to check the security of their device and configurations against the attacks described in the previous section. We have, first of all, tried to give an easy and intuitive interface. Moreover, the tool has been designed to be as portable as possible. To this aim we have decided to implement it in Java, via the IAIK PKCS#11 wrapper [4], which supports many existing devices.[2]

The attack window Figure 1 shows the attack windows of the tool where, using the buttons on the left, the user can perform six pre-existent attacks, plus custom attacks as described below. When an attack is successful, i.e., a sensitive key is extracted, the tool double-checks that the value of the extracted key coincides with the value of the key stored in the device: it encrypts the plaintext 'PKCS#11' under the stored key and then decrypts it with the extracted one, checking that the obtained plaintext is, again, 'PKCS#11'. The first two buttons on the right can be used to look for conflicting attributes on the keys stored in the device, and for performing a preliminary (static) test on the feasibility of the known attacks on such real keys. The third button, 'Attack test', tries to perform the six above mentioned attacks on the device real keys. The six buttons on the left, in fact, create ad-hoc keys for the specific attacks; while this is interesting for hardware developer, an administrator might only be interested in discovering weaknesses on the actual device configuration and keys and not just potential attacks on the subset of the standard the token implements. Button 'Find attack' produces the input for the two model checkers NuSMV and SATMC, via variations of the perl scripts described in [3, 5]: in order to increase the performance of the analysis, only the actual subset of PKCS#11 implemented on the token is analyzed.

The custom attack window Figure 2 shows the interface for custom attacks. This window allows users to perform all the typical operations performed during attacks: users can create keys, convert keys to byte streams and vice-versa, perform wrap/unwrap and encrypt/decrypt plus extra functions on a different windows, test extracted keys against stored ones. When creating new keys (see the windows on the right), it is possible to try to set their attributes. Since proprietary extensions of the standard restrict the setting of some attributes, the tool reports whether the requested setting has been successfully stored in the device.

Current and future work We are currently testing the tool on a number of commercial devices. So far, we have have found that many devices are indeed vulnerable to some or all of these attacks. We will publish more details after due notice has been given to the manufacturers in question. Meanwhile, the tool is still under development, and is partially supported by the MIUR Italian project SOFT: "*Security Oriented Formal Techniques*. In particular, we are making attack sequences more adaptable to the specific device; an attack may fail because one needed cryptographic primitive is not supported by the device, but a slight variant of the attack might still be possible using a different, supported, functionality. As a trivial example, think of using AES instead of DES. We are studying a graphical interface for custom attacks, so that attack logic can be more intuitively represented in terms of information flows from/to the device. We are extending the model-checking functionality so that the actual configuration, referring

[2] Compatibility has been reported with Giesecke & Devrient, Utimaco, Oberthur, SeTec, Orga, IBM, Safenet, Schlumberger, Gemplus, Dallas, Rainbow, ActivCard, A-Trust, A-Sign, Eracom, Aladdin, Mozilla, Eutron, TeleSec, nCipher.

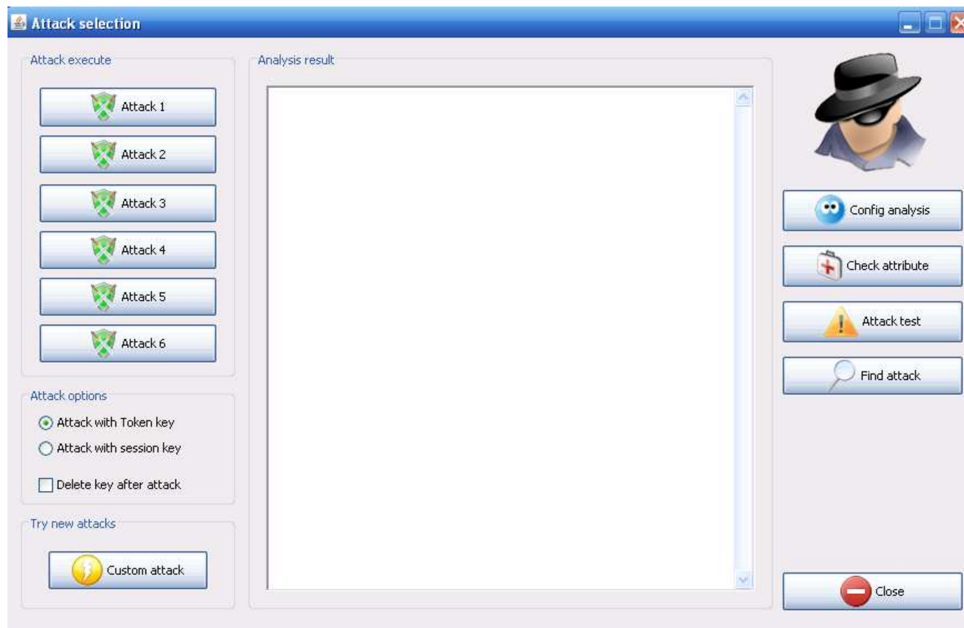

Figure 1: The attack window

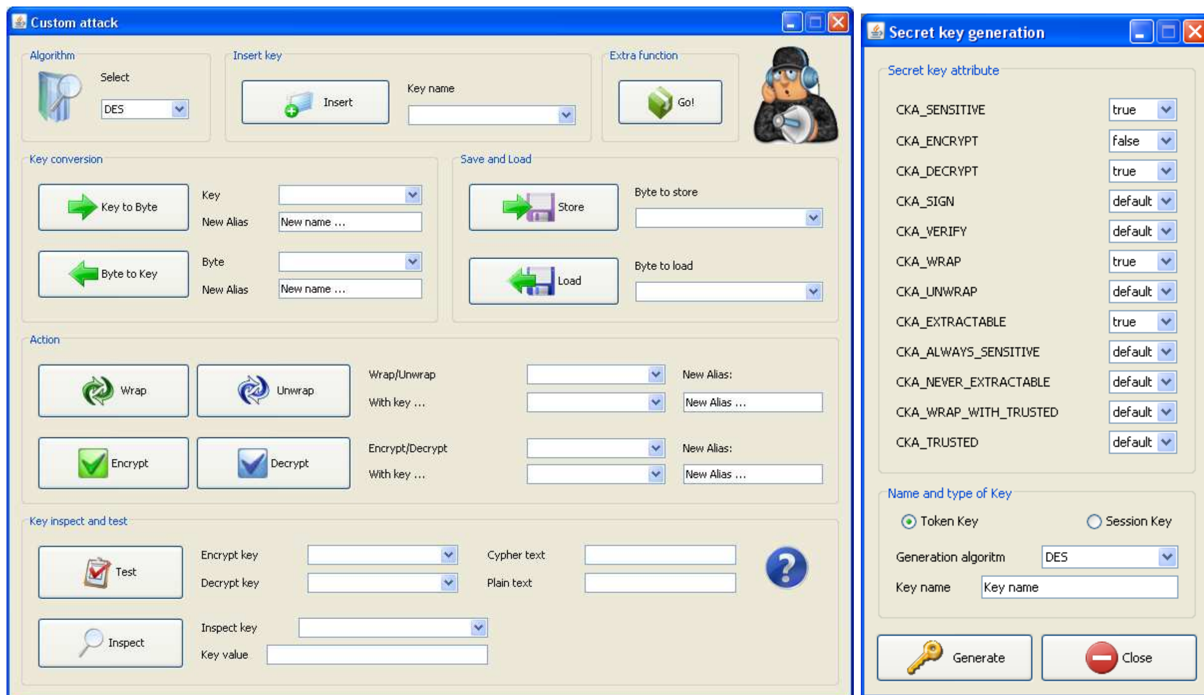

Figure 2: The custom attack window with key generation

to the real keys stored in the device, is included in the model. This might be useful to model-check a specific configuration of the device, instead of assuming the presence of weakly configured keys. We are also writing a parser for the output of model-checkers so that the theoretical attacks can be directly tested on the real devices. Finally, we intend to formalize the static analyses we already perform on the key attributes, to see if they can be used to statically validate specific device and configurations. This might complement the model checking analysis, by providing an additional tool for the static validation of real PKCS#11 devices.

References

[1] M. Bortolozzo and G. Marchetto. Vulnerabilità dello standard PKCS#11: dalla teoria alla pratica. Master's thesis, University of Venice, Italy, 2009.

[2] J. Clulow. The design and analysis of cryptographic APIs for security devices. Master's thesis, University of Natal, Durban, 2003.

[3] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, Pittsburgh, PA, USA, June 2008. IEEE Computer Society Press.

[4] Institute for Applied Information Processing and Communication (IAIK) of the Graz University of Technology. The IAIK Provider for the Java Cryptography Extension (IAIK-JCE) . `http://jce.iaik.tugraz.at/`.

[5] S. Fröschle and G. Steel. Analysing PKCS#11 Key Management APIs with Unbounded Fresh Data. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (ARSPA-WITS'09)*, 2009.

[6] J. Clulow. On the security of PKCS#11. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 411–425, 2003.

Monotonicity for Multiple Key Management Tokens

Johannes Borgström

Abstract

Formal analysis of cryptographic APIs such as PKCS #11 have focused on the case where the attacker only has access to a single token implementing the API. Operations that are non-monotonic in this simple setting, such as the setting of sticky attributes, may no longer be so in a multi-token setting. We show that this may lead to vulnerabilities, by giving an example attack where practices that are safe for a single token no longer are so for two communicating tokens. To our knowledge, this is the first multi-token attack on PKCS #11.

1 Introduction

The PKCS #11 standard specifies a class of cryptographic APIs with operations for key generation, encryption/decryption and key wrapping and unwrapping. We will here consider the case of an organization owning several cryptographic tokens, each implementing a PKCS #11-conforming API. These tokens are used for secure communication between users belonging to the organization, including key exchange.

One design goal of these tokens is to ensure the secrecy of keys that are used for key wrapping and/or unwrapping. In particular, it should not be possible for an adversarial user with access to the tokens only through their API to obtain such a key in plain text: if a wrapping key is compromised, then all past and future communications using keys wrapped under this key (transitively) are also affected.

As a mechanism to ensure secrecy, keys are decorated with properties, including which kinds of operations they may be used for. Critical properties are often *sticky*, meaning that they cannot be unset once set, and may *conflict* with other properties, in order to achieve separation of usage [2]. Setting a sticky property p on a key is intended to be a non-monotonic operation, in the sense that it restricts the attacker's future actions: the key can afterwards never be used for operations that conflict with the sticky property p. Local checks for non-monotonicity may no longer be globally effective in a system containing multiple tokens, as shown by the following attacks.

2 Multi-Token Attacks on PKCS #11

The PKCS #11 standard specifies an object model and interfaces to a number of functions, including cryptographic operations. In this paper, we focus on the interaction between encryption, decryption and key wrapping and unwrapping. For the sake of simplicity, we consider key objects as the only data stored within a token. We conservatively model the wrapping of one key with another as a normal encryption; this is motivated by key wrapping and unwrapping being built from normal encryptions and decryptions. Our treatment builds on Delaune et al. [3]. The attacks are along the lines of the pure API attacks described by Clulow [2].

The state of a cryptographic token is modelled as a finite map from abstract key handles $g,h \in \mathbf{H}$ to key objects. A key object contains a key $k,l \in \mathbf{K}$ and a bitmap specifying the properties of the key. In examples, we use the convention that h is a handle for a (wrapping) key k and g is the handle of a (data) key l. The properties that we consider are permissions to use the key for encryption, decryption, key wrapping and key unwrapping.

The set of permissions is $\mathbf{P} := \{\mathsf{enc}, \mathsf{dec}, \mathsf{wrap}, \mathsf{unwrap}\} \supseteq p,q$, and token states M thus have the type $\mathbf{H} \rightarrow^{\mathsf{fin}} \mathbf{K} \times 2^{\mathbf{P}}$. We often omit the braces when writing a permission set containing only one permission. We assume some primitive data objects $d \in \mathbf{D}$, and a Dolev-Yao style term algebra of shared-key cryptography with authenticated encryption $\mathrm{enc}(t,k)$ and decryption $\mathrm{dec}(t,k)$ using simple keys. The

constant symbols of the algebra are $\mathbf{H} \cup \mathbf{K} \cup \mathbf{D}$; we denote terms from the algebra by $t \in \mathbf{T}$. We write $\mathrm{consts}(t)$ for the set of constants occurring in t, and lift consts to sets of and relations over terms in the usual way.

We begin by modelling four operations supported by the token, intuitively described as follows.

1. $\texttt{enc}\ t\ \texttt{with}\ h$ means "output the encryption of input t with the key with handle h"
2. $\texttt{dec}\ \mathrm{enc}(t,k)\ \texttt{with}\ h$ means "output the decryption of input $\mathrm{enc}(t,k)$ with the key with handle h"
3. $\texttt{wrap}\ g\ \texttt{with}\ h$ means "output the key with handle g wrapped with the key with handle h"
4. $\texttt{unwrap}\ \mathrm{enc}(l,k)\ \texttt{with}\ h\ \texttt{for}\ q$ means "output a fresh handle with permissions q for a key unwrapped with the key with handle h from input $\mathrm{enc}(l,k)$"

We formalize the API operations as follows: given a token and some terms known to the user, an operation returns an updated token state and some additional terms that become known to the user, possibly creating fresh names. Note that unwrap is the only operation that generates fresh handles and/or modifies the token state (including the permissions of keys).

Specification of API Operations

$$
\begin{array}{rrcll}
\texttt{enc}\ t\ \texttt{with}\ h: & M,h,t & \rightarrow & M,\mathrm{enc}(t,k) & \text{if } M(h) = (k, p \cup \mathsf{enc}) \\
\texttt{dec}\ \mathrm{enc}(t,k)\ \texttt{with}\ h: & M,h,\mathrm{enc}(t,k) & \rightarrow & M,t & \text{if } M(h) = (k, p \cup \mathsf{dec}) \\
\texttt{wrap}\ g\ \texttt{with}\ h: & M,g,h & \rightarrow & M,\mathrm{enc}(l,k) & \text{if } M(h) = (k, p \cup \mathsf{wrap}) \text{ and } M(g) = (l,q) \\
\texttt{unwrap}\ \mathrm{enc}(l,k)\ \texttt{with}\ h\ \texttt{for}\ q: & & & & \\
 & M,h,\mathrm{enc}(l,k) & \xrightarrow{g} & M',g & \text{if } M(h) = (k, p \cup \mathsf{unwrap}) \\
 & & & & \text{and } M' = M \cup \{g \mapsto (l,q)\}
\end{array}
$$

Formally, the specification of an operation has type $(\mathbf{M} \times 2^{\mathbf{T}}) \times 2^{\mathbf{H} \cup \mathbf{K}} \times (\mathbf{M} \times 2^{\mathbf{T}})$. Informally, the specification $M,S \xrightarrow{H} M',S'$ means that a user that has access to a crypto token in state M and knows all terms in S can take the token into state M' and additionally learn the terms S', generating fresh handles (and later, keys) H. This is made precise by the following execution semantics. The execution relation operates on pairs of a multisets of states and a set of terms representing the adversary knowledge. Below, $\mathscr{M}$ is a multiset of tokens and $\uplus$ is multiset union.

Execution

$$\mathscr{M} \uplus M, S \cup S_0 \xrightarrow{H} \mathscr{M} \uplus M', S \cup S_0 \cup S' \quad \text{if} \quad M,S \xrightarrow{H} M',S' \text{ and } H \cap \mathrm{consts}(S \cup \mathrm{dom}(M)) = \emptyset$$

Here we clearly see the freshness condition on the generated data, and that the adversary knowledge is augmented by the terms learned.

2.1 An Attack on Two Tokens

We consider a network of two tokens, set up for the following scheme of interaction: The first token wraps a key and sends it to the second. The second token then unwraps the key and uses it to send encrypted data back to the first, which then can decrypt it. In our model, we can write the two tokens as $M_1 = \{h_1 \mapsto (k, \mathsf{wrap}), g_1 \mapsto (l, \mathsf{dec})\}$ and $M_2 = \{h_2 \mapsto (k, \mathsf{unwrap})\}$. The first token M_1 holds a key k, enabled for wrapping, and an decryption key l; the other token M_2 holds the same key k, enabled for unwrapping. We use the convention that h is a handle for k and g is a handle for l, with a subscript depending on which token the handle refers to.

A successful interaction takes place as shown below. For ease of reading, we show the transition rule that is used at each step in the left column. We write M_2' for $(M_2 \cup g_2 \mapsto (l, \mathsf{enc}))$ and let $S = \{h_1, h_2, g_1, d\}$ be the initial user/adversary knowledge.

Successful Interaction

		M_1, M_2, S
`wrap` g_1 `with` h_1 :	$\rightarrow$	$M_1, M_2, S \cup \{\mathsf{enc}(l,k)\}$
`unwrap` $\mathsf{enc}(l,k)$ `with` h_2 `for enc` :	$\xrightarrow{g_2}$	$M_1, (M_2 \cup g_2 \mapsto (l, \mathsf{enc})), S \cup \{\mathsf{enc}(l,k), g_2\}$
`enc` d `with` g_2 :	$\rightarrow$	$M_1, (M_2 \cup g_2 \mapsto (l, \mathsf{enc})), S \cup \{\mathsf{enc}(l,k), g_2, \mathsf{enc}(d,l)\}$
`dec` $\mathsf{enc}(d,l)$ `with` g_1 :	$\rightarrow$	$M_1, (M_2 \cup g_2 \mapsto (l, \mathsf{enc})), S \cup \{\mathsf{enc}(l,k), g_2, \mathsf{enc}(d,l)\}$

Starting at the same point as the previous example, an attack compromising the key k is as follows. We write $M_2' = M_2 \cup g_2 \mapsto (l, \mathsf{wrap})$ and omit the key handles in the adversary knowledge for clarity.

Attack 1

		M_1, M_2
`wrap` g_1 `with` h_1 :	$\rightarrow$	$M_1, M_2, \{\mathsf{enc}(l,k)\}$
`unwrap` $\mathsf{enc}(l,k)$ `with` h_2 `for wrap` :	$\xrightarrow{g_2}$	$M_1, (M_2 \cup g_2 \mapsto (l, \mathsf{wrap})), \{\mathsf{enc}(l,k)\}$
`wrap` h_2 `with` g_2 :	$\rightarrow$	$M_1, (M_2 \cup g_2 \mapsto (l, \mathsf{wrap})), \{\mathsf{enc}(l,k), \mathsf{enc}(k,l)\}$
`dec` $\mathsf{enc}(k,l)$ `with` g_1 :	$\rightarrow$	$M_1, (M_2 \cup g_2 \mapsto (l, \mathsf{wrap})), \{\mathsf{enc}(l,k), \mathsf{enc}(k,l), k\}$

Above, the first and last operations only are performed on token M_1. In contrast to the intended interaction, the user unwraps key l with `wrap` permissions, and uses it to wrap key k instead of encrypting d.

This example shows that local enforcement of separation of usage of keys can easily be circumvented given access to multiple related tokens. This arises from two issues: Individually secure token configurations may be insecure when composed; and the permissions of keys are not protected by the wrapping primitive. In the following, we tighten the latter property.

2.2 Exploiting Assumptions of Non-monotonicity

We additionally model API operations for generating and modifying permissions of keys. We model the operation to change the permissions associated with a handle as guarded by a *permission policy*, a binary predicate ϕ over $2^{\mathbf{P}}$. A user may change the permissions of a key from p to q iff $\phi(p,q)$ holds, and we consider a permission set p *valid* iff $\phi(p,p)$ holds. In a *well-formed* policy, $\phi(q,q)$ holds whenever $\phi(p,p)$ and $\phi(p,q)$ do. Key generation may only generate keys with valid permissions.

5. `adm` h `for` q means "set the permissions of handle h to q".
6. `gen for` p means "generate a key with permissions p".

Delaune et al.[3] describe a necessary permissions policy ϕ_d for avoiding key compromise. Informally, the policy is that all permissions (in $\mathbf{P}$) are sticky and that certain pairs of permissions can not both be in a valid permission set. Because of stickiness and conflicts, this policy is non-monotonous. Formally, $\phi_d(p,q) := p \subseteq q \wedge (\{\mathsf{enc}, \mathsf{unwrap}\} \not\subseteq q) \wedge (\{\mathsf{dec}, \mathsf{wrap}\} \not\subseteq q) \wedge (\{\mathsf{wrap}, \mathsf{unwrap}\} \not\subseteq q)$.

Additional API Operations

`adm` h `for` q :	$M \cup \{h \mapsto (k,p)\}, h$	$\rightarrow$	$M \cup \{h \mapsto (k,q)\}$	if $\phi_d(p,q)$
`gen for` p :	M	$\xrightarrow{h,k}$	$M \cup \{h \mapsto (k,p)\}, h$	if $\phi_d(p,p)$

The `adm` operation is the only operation that modifies the permissions of keys stored in the token.

Proprietary extensions to PKCS have been developed[1] to ensure that wrapped keys are always unwrapped with their permissions unchanged. Under this assumption, we exhibit the following attack, where $M_1 = \{h_1 \mapsto (k, \mathsf{wrap})\}$ and $M_2 = \{h_2 \mapsto (k, \mathsf{unwrap})\}$. As above, we do not show key handles in the adversary knowledge. We write $M_1' = (M_1 \cup \{g_1 \mapsto (l, \mathsf{wrap})\})$.

Attack 2

		$M_1, M_2, \{h_1, h_2\}$
`gen for` $\emptyset$:	$\xrightarrow{g_1, l}$	$M_1 \cup \{g_1 \mapsto (l, \emptyset)\}, M_2$
`wrap` g_1 `with` h_1 :	$\rightarrow$	$(M_1 \cup \{g_1 \mapsto (l, \emptyset)\}), M_2, \{\mathrm{enc}(l,k)\}$
`unwrap` $\mathrm{enc}(l,k)$ `with` h_2 `for` $\emptyset$:	$\xrightarrow{g_2}$	$(M_1 \cup \{g_1 \mapsto (l, \emptyset)\}), (M_2 \cup g_2 \mapsto (l, \emptyset)), \{\mathrm{enc}(l,k)\}$
`adm` g_1 `for wrap`	$\rightarrow$	$M_1', (M_2 \cup g_2 \mapsto (l, \emptyset)), \{\mathrm{enc}(l,k)\}$
`wrap` h_1 `with` g_1 :	$\rightarrow$	$M_1', (M_2 \cup g_2 \mapsto (l, \emptyset)), \{\mathrm{enc}(l,k), \mathrm{enc}(k,l)\}$
`adm` g_2 `for dec`	$\rightarrow$	$M_1', (M_2 \cup g_2 \mapsto (l, \mathsf{dec})), \{\mathrm{enc}(l,k), \mathrm{enc}(k,l)\}$
`dec` $\mathrm{enc}(k,l)$ `with` g_2 :	$\rightarrow$	$M_1', (M_2 \cup g_2 \mapsto (l, \mathsf{dec})), \{\mathrm{enc}(l,k), \mathrm{enc}(k,l), k\}$

This example shows that the possibility of adding permissions to a key after it has been exported or imported may fail to provide the intended separation properties, even when wrapping and unwrapping preserve permissions.

3 Conclusion

Based on the examples above, we conclude that multiple tokens should be explicitly taken into account when verifying cryptographic APIs. In particular, invariants that are intended to hold of the state of tokens should be composable, and enable realistic communication scenarios.

As part of this investigation, we have shown that the permissions policy described in [3] does not guarantee key secrecy in a multiple token scenario, even when key permissions are preserved by wrapping and unwrapping.

In ongoing work, we are implementing a key management API in F7 [1] and verifying secrecy properties through type-checking. Initial results suggest that an API and permission policy similar to the one considered in [4] can be proven secure against a symbolic attacker also in a multi-token setting.

Acknowledgments Thanks to Andy Gordon, Karthik Bhargavan and Mike Roe for helpful comments.

References

[1] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 17–32, 2008.

[2] J. Clulow. On the security of PKCS#11. In *Proceedings of CHES 2003*, pages 411–425, 2003.

[3] S. Delaune, S. Kremer, and G. Steel. Formal analysis of PKCS#11. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 331–344, 2008.

[4] S. Fröschle and G. Steel. Analysing PKCS#11 key management APIs with unbounded fresh data. In *Proceedings of ARSPA-WITS '09*, 2009. To Appear.

[1] These were recently studied in [4] in combination with a stronger permissions policy.